Automated Composition of Web Services by Planning at the Knowledge Level

M. Pistore
University of Trento
Via Sommarive 14
38050 Povo (TN), Italy
pistore@dit.unitn.it

A. Marconi and **P. Bertoli** and **P. Traverso**
ITC-IRST
Via Sommarive 18
38050 Povo (TN), Italy
[marconi, bertoli, traverso]@itc.it

Abstract

In this paper, we address the problem of the automated composition of web services by planning on their "knowledge level" models. We start from descriptions of web services in standard process modeling and execution languages, like BPEL4WS, and automatically translate them into a planning domain that models the interactions among services at the knowledge level. This allows us to avoid the explosion of the search space due to the usually large and possibly infinite ranges of data values that are exchanged among services, and thus to scale up the applicability of state-of-the-art techniques for the automated composition of web services. We present the theoretical framework, implement it, and provide an experimental evaluation that shows the practical advantage of our approach w.r.t. techniques that are not based on a knowledge-level representation.

1 Introduction

Research in planning is more and more focusing on the problem of the automated composition of web services: given a set of services that are published on the Web, and given a goal, generate a composition of the available services that satisfies the goal (see, e.g., [Narayanan and McIlraith, 2002]). In spite of the fact that several approaches have been proposed so far (see, e.g., [McIlraith and Son, 2002; Wu *et al.*, 2003; Sheshagiri *et al.*, 2003; Traverso and Pistore, 2004]), solving this problem *in practice*, by scaling up to realistic descriptions of web services, is far from trivial. Indeed it is widely recognized that web services must be modeled with nondeterministic and partially observable behaviors [Koehler and Srivastava, 2003; Hull *et al.*, 2003; Berardi *et al.*, 2003; McIlraith and Fadel, 2002; Traverso and Pistore, 2004; Martinez and Lesperance, 2004], and thus planning algorithms must work with incomplete information and with actions with uncertain effects. Moreover, in several application domains, web services cannot be simply modeled as atomic components, but as stateful processes whose interactions are intrinsically asynchronous [Fu *et al.*, 2004; Foster *et al.*, 2003; Pistore *et al.*, 2005].

Recent works address the problem of the practicality of the proposed solutions for web service composition. For instance, [Traverso and Pistore, 2004; Pistore *et al.*, 2005] propose a framework where web services are modeled with stateful, nondeterministic, and partially observable behaviors, and planning techniques based on symbolic model checking are used to address the scalability problem. However, these techniques work under the rather unrealistic assumption that web services can exchange only a very small number of data values. For instance, as the experimental results reported in [Traverso and Pistore, 2004; Pistore *et al.*, 2005] show, reasonable performances are obtained for web services whose variables can contain only two values. This amounts to say that amazon.com could sell just two books!

Luckily enough, as already stated in [Pistore *et al.*, 2005], the composition solution should not depend on the actual data exchanged among web services, in the same way as the operations that one has to perform to buy a book do not depend on the precise book one wants to buy. The flow of operations and interactions depends instead on whether the desired book is available or not, on whether its cost is affordable, and so on. The relevant issues for interacting with amazon.com are whether one *knows whether* the book is available or not, whether one *knows the value of* its price, and so on. The hope here is to apply *planning techniques at the knowledge level*, in the style of those proposed in [Petrick and Bacchus, 2002], or used in [McIlraith and Son, 2002; Martinez and Lesperance, 2004]. This would make it possible to model only those features of the services which are relevant to compose them, and thus allow for an efficient automated composition.

Unfortunately, applying knowledge-level planning to solve the automated composition problem presents a major difficulty. While in [Petrick and Bacchus, 2002] the planning domain at the knowledge level is defined by hand, this is impractical for web service automated composition tasks. The knowledge level domain must be extracted automatically from the description of the services that are published, e.g., in standard languages, like BPEL4WS [Andrews *et al.*, 2003]. The problem is therefore to devise a proper knowledge level model, which is suited for the automated composition task, and which can be obtained automatically from the published descriptions of the web services.

In this paper we propose a novel approach to the problem

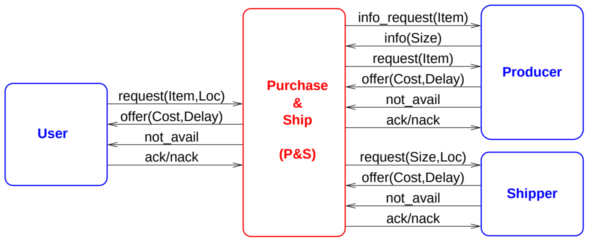

Figure 1: The Purchase & Ship example.

of automated composition that is based on planning at the knowledge level. We achieve this in the following steps:

- We formally define a knowledge level model of web services that can be obtained automatically from specifications in standard languages for modeling and implementing services. We show indeed how BPEL4WS processes can be automatically translated into their knowledge level models.
- We show how the automated composition problem can be described as a planning problem: from the knowledge level models of the available services and from the composition goal, we generate a planning problem such that a solution plan encodes the desired composition.
- We apply the planning techniques described in [Pistore *et al.*, 2005] to the knowledge level planning problem that we obtain from the previous steps. The key advantage w.r.t. [Pistore *et al.*, 2005] is that we do not have to deal with all the possible values of variables that are exchanged among services.
- We implement the proposed framework, and provide a preliminary experimental evaluation that clearly shows the benefits of our approach.

The paper is structured as follows. We first describe a simple example of composition of BPEL4WS processes, which we will use all along the paper (Section 2). We then briefly recall the framework for planning in asynchronous domains that has been first introduced in [Pistore *et al.*, 2005] (Section 3). In Section 4, we describe the knowledge level representation of web services, and the resulting planning framework at the knowledge level. We finally describe the experimental evaluation (Section 5), some conclusions, and related work.

2 Composition of BPEL processes

Our reference example is the Purchase and Ship (P&S hereafter) example introduced in [Pistore *et al.*, 2005; Traverso and Pistore, 2004].

Example 1 *The P&S example consists in providing a furniture purchase & ship service by combining two independent existing services, a furniture producer Producer and a delivery service Shipper. This way, the User, also described as a service, may directly ask the composite service P&S to purchase a given item and deliver it at a given place (for simplicity, we assume that the shipment origin is fixed and leave it implicit).*

The interactions with the existing services have to follow specific protocols. For instance, the interactions with the Shipper start with a request for transporting a product of a given size to a given location. This might not be possible, in which case the requester is notified, and the protocol terminates with failure. Otherwise, a cost and delivery time are computed and sent back to the requester. Then the Shipper waits for either an acceptance or a refusal of the offer from the invoker. In the former case, a delivery contract has been defined and the protocol terminates with success, while it terminates with failure in the latter case. Similar protocols are defined also for Producer and User. The messages exchanged among the involved services are described in Figure 1.

The P&S has the goal to sell home-delivered furniture (i.e., to reach the situation where the user has confirmed an order and the service has confirmed the corresponding suborders to producer and shipper), interacting with Shipper, Producer, and User according to their protocols. A typical interaction could be as follows:

1. *the User asks P&S for an article a, that he wants to be transported at location l;*
2. *P&S asks the Producer for the size, the cost, and how much time does it take to produce the article a;*
3. *P&S asks the Shipper for the price and time needed to transport an object of such a size to l;*
4. *P&S sends the User an offer which takes into account the overall cost (plus an added cost for P&S) and time to achieve its goal;*
5. *the User sends a confirmation of the order, which is dispatched by P&S to Shipper and Producer.*

This is however only the normal case, and other interactions should be considered, e.g., for the cases the producer and/or delivery services are not able to satisfy the request, or the user refuses the final offer.

With automated composition of web services we mean the generation of a new *composite service* (the P&S in our case) that interacts with a set of existing *component services* (Shipper, Producer, and User in our case) in order to satisfy a given *composition goal* (sell home-delivered furniture).

We assume that the interaction protocols of the component services, as well as the composite service, are described as BPEL4WS processes (see Figure 2 for a graphical representation of the Shipper BPEL4WS process). Notice that while existing services are described as abstract BPEL4WS processes (providing essentially the communication protocol), the synthesized service is an executable BPEL4WS process exporting all the details to be directly deployed and run. A BPEL4WS description specifies the types and internal variables of a service, and its input and output capabilities. The behavior of the process is described using input (`receive`) and output (`invoke`, `reply`) activities combined by standard constructs, such as sequences, loops, parallel executions, conditional choices, and nondeterministic choices.

From a BPEL4WS process, we can automatically extract a formal model of the interactions with the service, covering both the static aspects (e.g., its communication channels) and the behavioral aspects (defined in term of transition steps). For the moment, the translation is restricted to a

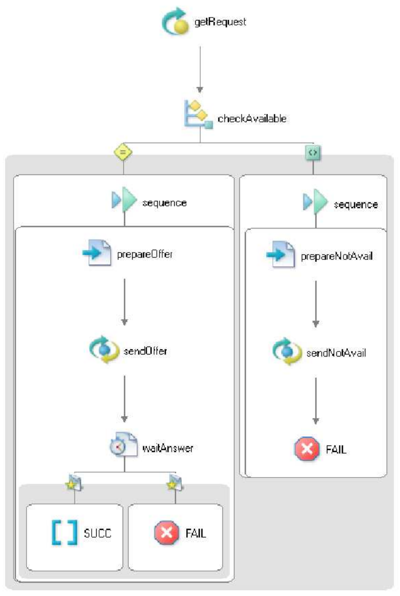

Figure 2: The Shipper BPEL4WS process.

subset of BPEL4WS processes: we support all BPEL4WS *basic* and *structured activities*, like `invoke`, `reply`, `receive`, `sequence`, `switch`, `while`, `flow` (without `links`) and `pick`; moreover we support `assignments` and a limited form of `correlation`. Our next steps will be dealing with `scopes` and with `fault`, `event` and `compensation handlers`.

For lack of space we omit the formal definition of this translation (details can be found at `http://astroproject.org/`) but we illustrate it in the case of the Shipper process.

Example 2 *Figure 3 shows a formal model of the Shipper, automatically extracted from the BPEL4WS process of Figure 2. The Shipper process is characterized by a set of abstract types used in the interactions (e.g.,* `Size`*,* `Cost`*), by a set of inputs and outputs (e.g.,* `request` *and* `offer`*, respectively), and by a set of typed functions used to manipulate internal variables (e.g., function* `costOf`*, which is used to obtain the cost of a particular shipping request). Variables are used to store information on the state of the process. In Figure 3, variables of abstract types (e.g.,* `customer_size`*) store values used in the communications, while an additional variable* `pc` *implements a "program counter" that holds the current execution step of the service. The model describes the evolution of the process through a set of possible transitions, each corresponding to a "step" in the BPEL4WS process; each transition defines an applicability condition, a firing action, and an effect (defined as a list of assignment to variables). Possible actions are inputs, outputs, and a special action TAU which is used to model internal evolutions of the process, such as assignments and decision making (e.g., the transition from state* `prepareOffer`*, where the internal functions* `costOf` *and* `delayOf` *are used to compute the shipping price and delivery time).*

```
PROCESS   Shipper;
TYPE
    Size;  Location;  Cost;  Delay;
INPUT
    request(Size, Location);  ack();  nack();
OUTPUT
    offer(Delay, Cost);  not_avail();
FUNC
    costOf(Size, Location) : Cost;  delayOf(Size, Location) : Delay;
VARIABLE
    pc: { START, getRequest, checkAvailable, end_checkAvailable, sequence1, sequence2,
         prepareOffer, sendOffer, waitAnswer, endWaitAnswer, empty1, prepareNotAvail,
         sendNotAvail, SUCC, FAIL };
    customer_size: Size;       customer_loc: Location;
    offer_delay: Delay;        offer_cost: Cost;
INIT
    pc := START;
TRANS
    pc = START  -[TAU]->  pc := getRequest;
    pc = getRequest  -[INPUT request(customer_size, customer_loc)]->  pc := checkAvailable,
    pc = checkAvailable  -[TAU]->  pc := sequence1;
    pc = checkAvailable  -[TAU]->  pc := sequence2;
    pc = sequence1  -[TAU]->  pc := prepareOffer;
    pc = prepareOffer  -[TAU]->  pc := sendOffer,
                                 offer_cost:=costOf(customer_size,customer_loc),
                                 offer_delay:=delayOf(customer_size,customer_loc);
    pc = sendOffer  -[OUTPUT offer(offer_cost, offer_delay)]->  pc := waitAnswer;
    pc = waitAnswer  -[INPUT nack]->  pc := FAIL;
    pc = waitAnswer  -[INPUT ack]->  pc := empty1;
    pc = empty1  -[TAU]->  pc := endWaitAnswer;
    pc = endWaitAnswer  -[TAU]->  pc := endCheckAvailable;
    pc = endCheckAvailable  -[TAU]->  pc := SUCC;
    pc = sequence2  -[TAU]->  pc := prepareNotAvail;
    pc = prepareNotAvail  -[TAU]->  pc := sendNotAvail;
    pc = sendNotAvail  -[OUTPUT not_avail]->  pc :=FAIL;
```

Figure 3: A formal model of the Shipper.

3 A Planning Framework for Service Composition

The work in [Pistore *et al.*, 2005] presents a formal framework for the automated composition of web services which is based on planning techniques: component services define the planning domain, composition requirements are formalized as a planning goal, and planning algorithms are used to generate the composite service. Due to the nature of web services, the resulting planning domain is nondeterministic and partially observable. The framework of [Pistore *et al.*, 2005] differs from other planning frameworks since it assumes an asynchronous, message-based interaction between the domain (encoding the component services) and the plan (encoding the composite service). More precisely, the planning domain is modeled as a *state transition system* (STS from now on) that can be in one of its possible states (a subset of which are *initial*) and can evolve to new states as a result of performing some actions. In particular, *input* actions represent messages sent to the component services, while *output* actions are messages received from the component services. *Private* actions are actions that the composite service can perform internally, without interacting with the component services[1], while the special action τ is used to model internal evolutions of the component services which are not visible to the service user. Finally, a labeling function associates to each state the set of properties $\mathcal{P}rop$ holding in that state.

[1]As we will see, private actions are used to model operations such as computing the total cost for the user from the costs received from the producer and from the shipper. Private actions are specific of the knowledge-level approach presented here and for this reason they do not appear in [Pistore *et al.*, 2005]. The extension of the theory of [Pistore *et al.*, 2005] to private actions is straightforward.

Definition 1 (state transition system (STS))
A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{L} \rangle$ *where:*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{I}$ *is the finite set of input actions;*
- $\mathcal{O}$ *is the finite set of output actions;*
- $\mathcal{A}$ *is the finite set of private actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \mathcal{A} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation;*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}rop}$ *is the labeling function.*

In a composition problem, the composite service is defined as a "controller" Σ_c (also described as a STS), which interacts with the domain Σ, orchestrating the component services by invoking their operations and handling results. We now recall the formal definition of the behavior of a STS Σ when controlled by Σ_c.

Definition 2 (controlled system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{L} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{A}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ *be two state transition systems, where* $\mathcal{L}_\emptyset(s_c) = \emptyset$ *for all* $s_c \in \mathcal{S}_c$. *The STS* $\Sigma_c \triangleright \Sigma$, *describing the behaviors of system* Σ *when controlled by* Σ_c, *is defined as:*
$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{L} \rangle$
where:

- $\langle (s_c, s), \tau, (s_c', s) \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s_c' \rangle \in \mathcal{R}_c$;
- $\langle (s_c, s), \tau, (s_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s' \rangle \in \mathcal{R}$;
- $\langle (s_c, s), a, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$, *with* $a \neq \tau$, *if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *and* $\langle s, a, s' \rangle \in \mathcal{R}$.

Due to the asynchronous nature of web service interactions, and in order to guarantee a correct execution of the composite service, we need to rule out explicitly the cases where the sender is ready to send a message that the receiver is not able to accept. According to [Pistore *et al.*, 2005], a state s is able to accept a message a if there exists some successor s' of s, reachable from s through a (possibly empty) sequence of τ transitions, such that an input transition labeled with a can be performed in s'. This intuition is captured in the following definition, where we denote by τ-closure(s) the set of states reachable from s through a chain of τ transitions. For what concerns private actions, since they correspond to internal operations of the composite service, we simply assume that they are executable in the current state of Σ.

Definition 3 (deadlock-free controller)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{L} \rangle$ *be a STS and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{A}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ *be a controller for* Σ. Σ_c *is said to be* deadlock free *for* Σ *if all states* $(s_c, s) \in \mathcal{S}_c \times \mathcal{S}$ *that are reachable from the initial states of* $\Sigma_c \triangleright \Sigma$ *satisfy the following conditions:*

- *if* $\langle s, a, s' \rangle \in \mathcal{R}$ *with* $a \in \mathcal{O}$ *then there is some* $s_c' \in \tau$*-closure*(s_c) *such that* $\langle s_c', a, s_c'' \rangle \in \mathcal{R}_c$ *for some* $s_c'' \in \mathcal{S}_c$;
- *if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *with* $a \in \mathcal{I}$ *then there is some* $s' \in \tau$*-closure*(s) *such that* $\langle s', a, s'' \rangle \in \mathcal{R}$ *for some* $s'' \in \mathcal{S}$.
- *if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *with* $a \in \mathcal{A}$ *then* $\langle s, a, s' \rangle \in \mathcal{R}$ *for some* $s' \in \mathcal{S}$.

In [Pistore *et al.*, 2005], the composition problem for domain Σ and composition goal ρ is defined as the problem of finding a deadlock-free STS Σ_c such that $\Sigma_c \triangleright \Sigma$ satisfies ρ. Planning techniques, based on the *planning as model checking* framework, are used to solve this problem, and the experimental results show the effectiveness of this approach.

However, the approach of [Pistore *et al.*, 2005] requires a finite set of data values for all data types (e.g., only a finite number of articles, locations, costs... can be defined in the P&S domain). This assumption guarantees that the formal model of a BPEL4WS process such as the one in Figure 3 can be mapped into a finite-state STS. Moreover, the number of data values impact dramatically on the size of the generated STS, and very small sets of values need to be used to allow for an effective plan generation: for instance, all experiments in [Pistore *et al.*, 2005] assume only two values for each data type. In the following we show how to adapt the framework of [Pistore *et al.*, 2005] to a knowledge-level planning approach in order to remove this restriction and allow for automated composition with a realistic (or even infinite) number of data values.

4 Service Composition via Knowledge-Level Planning

The key aspect for extending the approach of [Pistore *et al.*, 2005] to the knowledge level is the definition of an appropriate model for providing a knowledge level description of the component services. In this section we formally define such a model in terms of a suitable knowledge base (from now on KB). Then we show how to construct the STS corresponding to the planning domain by composing the knowledge bases of the component services.

Definition 4 (Knowledge Base)
A knowledge base KB *is a set of propositions of the following form:*

- $K_V(x)$ *where* x *is a variable with an abstract type;*
- $K(x = v)$ *where x is an enumerative variable and v is one of its possible values;*
- $K(x = y)$ *where x and y are two variables with the same type;*
- $K(x = f(y_1, \ldots, y_n))$ *where* $x, y_1, \ldots, y_n$ *are variables with an abstract type and* f *is a function compatible with the types of* $x, y_1, \ldots, y_n$.

With $K(p)$ we mean that we know that proposition p is true and with $K_V(x)$ that we know the value of the variable x. This definition of knowledge base is very simple; still, our experiments show that it is powerful enough to model web service composition problems.

We say that a knowledge base KB is *consistent* if it does not contain contradictory knowledge propositions such as $K(x = v)$ and $K(x = v')$, with $v \neq v'$. We say that KB is *closed under deduction* if it contains all the propositions that can be deduced from the propositions in KB; for instance a

KB containing both $K_V(x)$ and $K(x = y)$ should contain also $K_V(y)$.

The knowledge base KB of a component service is obtained from the variables, functions and types of the service.

Example 3 *An example of* knowledge base *for the Shipper process in Figure 3 is:*

```
KB={K(pc = waitAnswer),K_V(customer_size),K_V(customer_loc),
  K(offer_cost = costOf(customer_size, customer_loc)),
  K(offer_delay = delayOf(customer_size, customer_loc)),
  K_V(offer_cost), K_V(offer_delay)}.
```

In the following we describe when a transition can be executed in a knowledge base KB and how its execution affects KB.

We model a transition t, such as those presented in Figure 3, as a triple (C, a, E) where $C = (c_1 \wedge \ldots \wedge c_n)$ are its conditions, a is its firing action and $E = (e_1; \ldots; e_n)$ are its effects. We start by defining the auxiliary *restriction* and *update* operations.

The *restriction* of a knowledge base KB with a condition C, denoted with $restrict(KB, C)$, is performed adding to KB the knowledge obtained from C and closing under deduction; for instance:

```
restrict({K(x = y)}, y = z) =
  {K(x = y), K(y = z), K(x = z)}.
```

The *update* of a knowledge base KB with an effect E, denoted with $update(KB, E)$, consists in performing the following steps: for each assignment in E, remove from KB the knowledge we had on the modified variable, add the knowledge derived from the assignment and close the KB under deduction. For instance:

```
update({K(x = y)}, z := x; x := w) =
  update({K(x = y), K(z = x), K(z = y)}, x := w) =
  {K(z = y), K(x = w)}.
```

We now give the definitions of *applicability* and *execution* which depend on a service transition t and on an action a_c performed by a peer interacting with the service; the firing action a in t and the peer action a_c correspond to the same action except that the former is instantiated on service variables, while the latter on variables of the peer. Consider for instance the following example where action `request(s, l)` performed by the peer corresponds to the Shipper `request(customer_size, customer_loc)`.

Example 4 *Let's consider the input transition* `t` *of the Shipper*

```
pc = getRequest
 -[INPUT request(customer_size, customer_loc)]->
pc := checkAvailable
```

and suppose that our current knowledge base is:

```
KB = { K(pc = getRequest), K_V(s), K_V(l) } .
```

where `s:Size` *and* `l:Location` *are additional variables. Action* `request(s, l)`*, corresponding to invoking the Shipper action using* `s` *and* `l` *as parameters, is applicable in* `KB` *since we know the values of* `s` *and* `l` *and the condition of transition* `t` *is obviously consistent with* `KB`.

The knowledge base obtained by executing on `KB` *transition* `t` *and action* `request(s, l)`*, is defined as follows:*

- *we restrict* `KB` *with the knowledge* `K(pc = getRequest)` *associated to the transition condition; in this specific case* `KB` *remains unchanged;*
- *then we update* `KB` *with the knowledge carried by the input action on the variables used as action parameters; this consists in removing from* `KB` *all the knowledge we had on* `customer_size` *and* `customer_loc`*, adding the new knowledge* `K(customer_size = s)`, `K(customer_loc = l)` *and closing under deduction; we obtain:*

```
KB' = { K(pc = getRequest), K_V(s),
  K_V(l), K_V(customer_size), K_V(customer_loc),
  K(customer_size = s), K(customer_loc = l) } ;
```

- *finally we update the knowledge base with the effects, removing from* `KB'` *all the knowledge we had on the variables modified by the assignments, adding the new knowledge (in this case* `K(pc := checkAvailable)`*) and closing under deduction; we obtain:*

```
KB'' = { K(pc = checkAvailable), K_V(s),
  K_V(l), K_V(customer_size), K_V(customer_loc),
  K(customer_size = s), K(customer_loc = l) } .
```

Definition 5 (KL Applicability and Execution) *A transition t=(C, a, E) and a corresponding action a_c are* applicable *in KB, written $applicable[t](KB, a_c)$ if:*

- *a_c=$i(x_1, \ldots, x_n)$ and a=$i(y_1, \ldots, y_n)$, where i is an input, $K_V(x_1), \ldots, K_V(x_n) \in KB$ and $restrict(KB, C)$ is consistent; or*
- *a_c=$o(x_1, \ldots, x_n)$ and a=$o(y_1, \ldots, y_n)$, where o is an output, and $restrict(KB, C)$ is consistent; or*
- *a_c=a=τ and $restrict(KB, C)$ is consistent.*

If $applicable[t](KB, a_c)$, then we denote with KB'=$exec[t](KB, a_c)$ the execution *on KB of t and a_c, defined as follows:*

- *if a_c=τ and t=(C, τ, E) then KB'=$update(KB'', E)$, where KB''=$restrict(KB, C)$;*
- *if a_c=$i(x_1, \ldots, x_n)$, where i is an input, and t=$(C, i(y_1, \ldots, y_n), E)$ then KB'=$update(KB'', y_1$:=$x_1; \ldots; y_n$:=$x_n; E)$, where KB''=$restrict(KB, C)$;*
- *if a_c=$o(x_1, \ldots, x_n)$, where o is an output, and t=$(C, o(y_1, \ldots, y_n), E)$ then KB'=$update(KB'', x_1$:=$y_1; \ldots; x_n$:=$y_n; E)$, where KB''=$restrict(KB, C) \cup \{K_V(y_1), \ldots, K_V(y_n)\}$.*

Notice that the execution of a transition increases the knowledge in KB, not only with the information in the effects, but also with those in the condition. Indeed, if the transition is executed, this means that the condition is known to hold.

The planning domain at the knowledge level is constructed from the knowledge-level models of each component service and from a knowledge-level representation of the composition goal. The latter defines which are the variables and the functions of the composite service, like, for instance, the cost of the offer to the user, or a special function that adds a mark up to the sum of the costs of the shipper and producer. We call these variables and functions, *goal variables* and *goal functions*.

Given a composition goal, we automatically generate its knowledge level representation that declares what the composite service must know and how goal variables and functions must be related with the variables and functions of the component services.

Example 5 *A possible (very simple) composition goal for our reference example is:*

```
TryReach
  user.pc = SUCC ∧ producer.pc = SUCC ∧ shipper.pc = SUCC ∧
  user.offer_cost =
    addCost(producer.costOf(user.article),
    shipper.costOf(producer.sizeOf(user.article), user.location))
```

The goal declares that we want all the services to reach the situation where the order has been confirmed. Moreover it states that the offered cost must be obtained by applying the function `addCost` *to the costs offered by the producer and the shipper. The operator* ***TryReach*** *is one of the modal operators provided by the* EAGLE *goal specification language. It requires that the plan reaches the goal condition whenever possible in the domain. For further details, see [Pistore* et al.*, 2005]*

To obtain the knowledge level goal, we flatten the functions introducing auxiliary variables until only basic propositions are left:

```
TryReach
  user.pc = SUCC ∧ producer.pc = SUCC ∧ shipper.pc = SUCC ∧
  user.offer_cost = goal.added_cost ∧
  goal.added_cost = addCost(goal.prod_cost, goal.ship_cost) ∧
  goal.prod_cost = producer.costOf(goal.user_art) ∧
  goal.ship_cost = shipper.costOf(goal.prod_size, goal.user_art) ∧
  goal.user_art = user.article ∧
  goal.prod_size = producer.sizeOf(goal.user_art) ∧
  goal.user_loc = user.location
```

From this flattened goal we can extract the goal variables: `goal.added_cost`, `goal.prod_cost`, `goal.ship_cost`, `goal.prod_size`, `goal.user_art`, *and* `goal.user_loc`*; and the goal function* `goal.addCost(Cost, Cost):Cost`.

The goal can then be automatically translated into its corresponding knowledge level goal:

```
TryReach
 K(user.pc = SUCC) ∧ K(producer.pc = SUCC) ∧ K(shipper.pc = SUCC) ∧
 K(user.offer_cost = goal.added_cost)∧
 K(goal.added_cost = goal.addCost(goal.prod_cost, goal.ship_cost))∧
 K(goal.prod_cost = producer.costOf(goal.user_art))∧
 K(goal.ship_cost = shipper.costOf(goal.prod_size, goal.user_art))∧
 K(goal.user_art = user.article)∧
 K(goal.prod_size = producer.sizeOf(goal.user_art))∧
 K(goal.user_loc = user.location)
```

The knowledge-level representation of the composition goal defines therefore a further knowledge base, that we call the *knowledge base of the goal*.

The knowledge-level planning domain Σ is obtained by combining the knowledge bases of the component services and the knowledge base of the goal, by instantiating the input and output actions of the component services on goal variables, and by adding the private actions obtained by applying goal functions to goal variables.

Definition 6 (Knowledge-level Planning Domain) *The planning domain Σ for a composition problem is an STS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{L} \rangle$ defined as follows:*

- *the set of states $\mathcal{S}$ are all the possible KB defined on the set of typed variables $X = \bigcup_{i=0\ldots n} X_i$ and on the set of typed functions $F = \bigcup_{i=0\ldots n} F_i$, where $X_1, \ldots, X_n$ and $F_1, \ldots, F_n$ are the variables and functions of the component services, while X_0 and F_0 are those of the composition goal;*
- *$\mathcal{S}^0 \subseteq \mathcal{S}$ is the set of initial states corresponding to the initial knowledge bases $KB_0^1, \ldots, KB_0^n$, obtained from the initial assignments of the component services;*
- *$\mathcal{I}$ is the set of input actions $i(x_1, \ldots, x_n)$ such that $i(y_1, \ldots, y_n)$ is an input action in a transition of a component service and $x_1, \ldots, x_n$ are goal variables with the same type of service variables $y_1, \ldots, y_n$;*
- *$\mathcal{O}$ is the set of output actions $o(x_1, \ldots, x_n)$ such that $o(y_1, \ldots, y_n)$ is an output action in a transition of a component service and $x_1, \ldots, x_n$ are goal variables with the same type of service variables $y_1, \ldots, y_n$;*
- *$\mathcal{A}$ is the set of private actions $x_0 := f(x_1, \ldots, x_n)$ where f is a goal function and $x_0, x_1, \ldots, x_n$ are goal variables compatible with the type of f;*
- *$\mathcal{R}$ is the set of transitions $r = \langle s, a_c, s' \rangle$, with $s, s' \in \mathcal{S}$, such that:*
  - *if a_c is an input, output or τ action, then there exists a $t = (C, a, E)$ in the sets of transitions of the component services such that $applicable[t](s, a_c)$ and $s' = exec[t](s, a_c)$;*
  - *if a_c is a private action of the form $x_0 := f(x_1, \ldots, x_n)$, then $K_V(x_1), \ldots, K_V(x_n) \in s$ and $s' = update(s, x_0 := f(x_1, \ldots, x_n))$;*
- *$\mathcal{L}$ is the trivial function associating to each state the set of propositions that hold in that state.*

Given the domain Σ described above, we can apply the approach presented in [Pistore *et al.*, 2005] and obtain a deadlock free Σ_c that controls Σ by satisfying the composition requirements ρ. Despite of the fact that the synthesized controller Σ_c is modelled at the knowledge level, its elementary actions model communication with the component services (sending and receiving of messages) and manipulation of goal variables; given this, it is straightforward to obtain the executable BPEL4WS composite service from Σ_c.

5 Experimental Evaluation

We have implemented the proposed approach, and used the planning as symbolic model checking techniques presented in [Pistore *et al.*, 2005] to perform an experimental evaluation. All the tests have run on a 3 GHz Xeon PC, limiting memory usage to 512MBytes, and with a CPU timeout of 1000 seconds.

We first considered the P&S example explained in the previous sections, which, in spite of the reduced number of components, requires a rather intricated protocol to be established for achieving the goal. Figure 4 shows the results of our

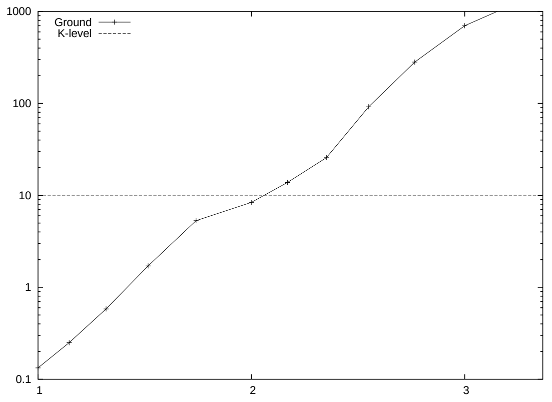

Figure 4: Experiments with P&S.

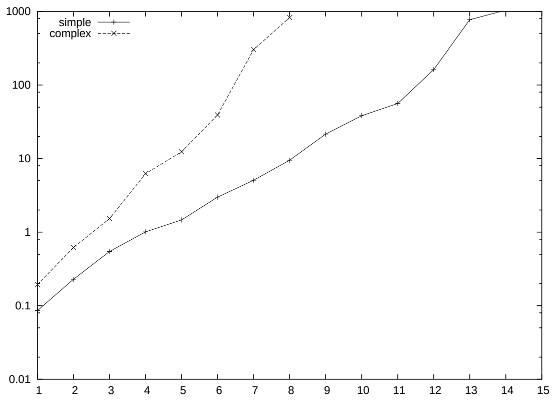

Figure 5: Experiments with parametrized domains.

experiments. The same planning engine is run to perform knowledge-level composition and ground level compositions, where different ranges of values can be produced and exchanged by the Shipper, Producer and User. The horizontal axis reports the cardinality $\overline{n}$ of the data types (i.e. Size, Location, Cost, Delay) handled by the services. We also consider intermediate cases where we have, for instance, two possible values for the Cost and only one value for the Size), reporting the average cardinality in the figure. On the vertical axis, we report the composition time. As expected, ground level composition is only feasible for the unrealistic cases where processes may exchange only data with 2 or at most 3 values. Indeed, the time for ground composition grows exponentially with the cardinality of the data types, and even the simple case where types have cardinality 4 is unmanageable. On the contrary, knowledge-level composition takes about 10 seconds to complete, with a performance similar to that of the ground level for $\overline{n} = 2$. This is a reasonable result, since, basically, binary variables at the ground level correspond to (binary) knowledge atoms at the knowledge-level.

To evaluate the scalability of the knowledge-level approach when the number of component services grows, we perform two sets of experiments, considering a generalization of the example domain that involves a set of component services. In our first set of experiments, each component is represented by a very simple abstract BPEL4WS process that is requested to provide a service and can respond either by performing the service, or by refusing. The composition requirement is that either all services end successfully, or a failure is reported to the invoker of the composed service. Figure 5 reports the knowledge-level composition times, for increasing number of services to be composed (indicated on the horizontal axis). The composition achieves results comparable to those reported in [Pistore *et al.*, 2005], where ground composition is performed only considering the case of types with range of cardinality 2. We manage to compose 14 services in 20 minutes.

In the second set of experiments, also reported in Figure 5, we make the protocol more complex, by requiring a higher degree of interleaving between components. Here, the interactions with each component are more complex than a single invoke-response step, and, to achieve the goal, it is necessary to carry out interactions with all components in an interleaved way. Such interleaving is common in the P&S example where, e.g., the P&S cannot confirm the order to the producer if shipping is not available or if the user does not accept the offer. The increased complexity reflects on the complexity of the composition; however, compositions of reasonable complexity (up to 8 services) can be achieved within 20 minutes. Even in this case, automated composition takes a rather low amount of time, surely faster than manual development of BPEL4WS code.

6 Conclusions and Related Work

In this paper, we have shown how automated composition can be obtained by translating automatically process-level descriptions of web services, e.g., BPEL4WS processes, to a planning problem that describes the interactions among services at the knowledge level. Our solution exploits the framework for planning in asynchronous domains presented in [Pistore *et al.*, 2005]. We show experimentally how the knowledge level approach increases significantly the practical applicability of this framework.

The idea of planning at the knowledge level is not new, and the solution that we propose is based on the idea presented in [Petrick and Bacchus, 2002]. Notice however that our work addresses an orthogonal problem, i.e., the problem of how a knowledge-level planning domain can be automatically generated from a set of BPEL4WS processes that describe web services. Moreover, at the technical level, our works differs from those proposed in [Petrick and Bacchus, 2002] in the kind of information that we represent and store in the knowledge bases, as well as in the knowledge-level planning domain that we automatically generate, and in the planning algorithm we exploit.

The work in [Martinez and Lesperance, 2004] shows how to use the knowledge level planner presented in [Petrick and Bacchus, 2002] to solve service composition problems. However they deal only with atomic services (services exporting a single operation instead of a protocol) and the knowledge level domain (e.g. domain action specification, update rules) is written by hand.

Different automated planning techniques have been proposed to tackle the problem of service composition, see, e.g., [Wu *et al.*, 2003; Sheshagiri *et al.*, 2003]. However, most of them cannot deal with the problem that we address in this paper, since they cannot deal with conditional outputs, uncertain action effects, and partial observability. A remarkable exception is the work described in [Narayanan and McIlraith, 2002; McIlraith and Son, 2002; McIlraith and Fadel, 2002], which instead deals with sensing actions and knowledge level predicates. However, even in this work, the knowledge level domain is given by hand, and the problem of devising a knowledge base that can be generated automatically from BPEL4WS web services descriptions is not addressed. Moreover, the composition problem is limited to sequential compositions of services.

The work in [Hull *et al.*, 2003] presents a formal framework for composing e-services from behavioral descriptions given in terms of automata; they focus on the theoretical foundations, without providing practical implementations. Moreover, the considered e-composition problem is fundamentally different from our process-level composition, since it is seen as the problem of generating a set of rules coordinating the execution of the available services. No concrete and executable process is generated with that approach. This is the main difference also with the work described in [Berardi *et al.*, 2003].

In the future, we plan to extend the work to the automated composition of semantic web services, e.g., described in OWL-S [OWL-S, 2003], along the lines of the work in [Traverso and Pistore, 2004], which at the moment is not based on knowledge-level planning. We will also explore variants of our knowledge base, so to allow flexible ways to model different features of ground level domains.

Acknowledgements This work is partially funded by the MIUR-FIRB project RBNE0195K5, "Knowledge Level Automated Software Engineering", and by the MIUR-PRIN 2004 project "Advanced Artificial Intelligence Systems for Web Services". The authors want to thank all members of the Astro project (`http://astroproject.org/`) for their collaboration and their feedback.

References

[Andrews *et al.*, 2003] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[Berardi *et al.*, 2003] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-Services that Export their Behaviour. In *Proc. ICSOC'03*, 2003.

[Foster *et al.*, 2003] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. ASE'03*, 2003.

[Fu *et al.*, 2004] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW'04*, 2004.

[Hull *et al.*, 2003] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.

[Koehler and Srivastava, 2003] J. Koehler and B. Srivastava. Web Service Composition: Current Solutions and Open Problems. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*, 2003.

[Martinez and Lesperance, 2004] E. Martinez and Y. Lesperance. Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning. In *Proc. of ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[McIlraith and Fadel, 2002] S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. NMR'02*, 2002.

[McIlraith and Son, 2002] S. McIlraith and S. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. KR'02*, 2002.

[Narayanan and McIlraith, 2002] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.

[OWL-S, 2003] OWL-S. OWL-S: Semantic Markup for Web Services (OWL-S version 1.0), 2003.

[Petrick and Bacchus, 2002] R. Petrick and F. Bacchus. A Knowledge-Based Approach to Planning with Incomplete Information and Sensing. In *Proc. AIPS'02*, 2002.

[Pistore *et al.*, 2005] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[Sheshagiri *et al.*, 2003] M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.

[Traverso and Pistore, 2004] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. ISWC'04*, 2004.

[Wu *et al.*, 2003] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.